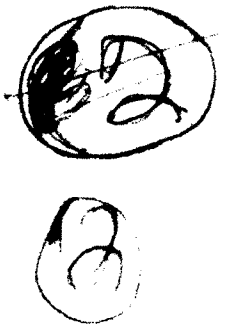

University of Southern California Information Sciences Institute Common Lisp Framework Final Technical Report

David S. Wile

Sponsored by Defense Advanced Projects Agency (DOD)
DARPA/CSTO
Title of Contract:
"A Proposed Research Program in Strategic Computing"
Project: Common LISP Framework
DARPA Order No. 6096
Issued by DSS-W under Contract MDA903-87-C-0641
Period of Performance: 09/01/87 - 05/31/92

February 8, 1993

[1]The views and conclusions contained in this document must not be interpreted as representing the official policies either expressed or implied of the Defense Advanced Research Projects Agency or the U.S. Government.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: USC INFORMATION SCIENCES INSTITUTE	6b. OFFICE SYMBOL
7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State, and ZIP Code): 4676 Admiralty Way, Marina del Rey, California 90292	7b. ADDRESS (City, State, and Zip Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: ~~Office of Naval Research~~ ARPA	8b. OFFICE SYMBOL
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: MDA903-87-C-0641	
8c. ADDRESS (City, State, and ZIP Code): University of California at San Diego (A-034), Scripps Institute of Oceanography, 8603 LaJolla Shores Drive, San Diego, CA. 92093-0234	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (INCLUDE SECURITY CLASSIFICATION)

Common Lisp Framework

12. PERSONAL AUTHOR(S)

David S. Wile

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 9/01/87 TO 5/31/92	February 8, 1993	20

16. SUPPLEMENTARY NOTATION

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)

Programming Environments, Persistent Objects, Relational Abstraction, Program Generators, Codewalkers, Transformational Programming, Program Specification, User Interfaces

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The principle goal of the COMMON LISP Framework project was to support Strategic Computing (SC) contractors with a comprehensive, state-of-the-art programming framework for the development and evolution of COMMON LISP programs. The CLF has affected the technical community by shaping both the structure of modern programming environments and the development paradigm they support. This structure rests upon an active objectbase which maintains integrity, integrated tools and provides persistence. CLF's integrity management is unique in providing declarative integrity condition statements--which can be reasoned about and analyzed--while still permitting efficient algorithmic maintenance of these conditions. Its declarative persistence mechanism, based on conceptual aggregations of objects, is also unique. Tools are integrated with the objectbase rather than each other. They gather their inputs from, and place their results back in, the objectbase and are invoked (demonically) by relative activity in the objectbase.

The final years of the COMMON LISP framework project were primarily devoted to extracting important functionality from the CLF and packaging it in a form suitable for use in other environments. This allowed its use by clients who could not use the monolithic CLF system in their in place system architectures.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b TELEPHONE (Include Area Code)	22c OFFICE SYMBOL
	(310) 822-1511	

DD FORM 1473, 92 JAN

SECURITY CLASSIFICATION OF THIS PAGE

1 Task Objectives

The principal goal of the COMMON LISP Framework project was to support Strategic Computing (SC) contractors with a comprehensive, state-of-the-art programming framework for the development and evolution of COMMON LISP programs. This framework provides the programmer with a fileless environment of persistent objects. Their relationships with other objects are manipulated via a set of generic operations and are used to associatively retrieve the objects. The COMMON LISP Framework system (the CLF) maintains the consistency of these objects and initiates automated processing for the programmer via a set of rules.

The CLF has affected the technical community by shaping both the structure of modern programming environments and the development paradigm they support. This structure rests upon an active objectbase which maintains integrity, integrates tools, and provides persistence. CLF's integrity management is unique in providing declarative integrity condition statements–which can be reasoned about and analyzed–while still permitting efficient algorithmic maintenance of those conditions. Its declarative persistence mechanism, based on conceptual aggregations of objects, is also unique. The persistence of these aggregates is automatically maintained. Tools are integrated with the objectbase rather than each other. They gather their inputs from, and place their results back in, the objectbase and are invoked (demonically) by relevant activity in the objectbase. Military and societal impact will accrue from the infusion into routine practice of programming environments which maintain all information relevant to the programming process within them; viz., product lifetime support, synergistic tool support and tightly coupled systems.

The final years of the COMMON LISP Framework project – covered in this report – were primarily devoted to two goals, one mundane and the other, laudable. First, we needed to port the CLF to our Hewlett Packard 400 Series workstations as the underpinnings for our everyday computing environment. More importantly, we wanted to open up our technology to other researchers by extracting important functionality from the CLF and packaging it in a form suitable for use in other environments. This allowed use of the functionality without potential clients having to buy off on the monolithic CLF system, which often could not be used in their in-place system architectures.

2 Technical Problem

The idea for the COMMON LISP Framework project arose as DARPA recognized the importance of the ongoing research in the Formalized System Development project at ISI, especially its potential for use by its SC contractors. The project began in

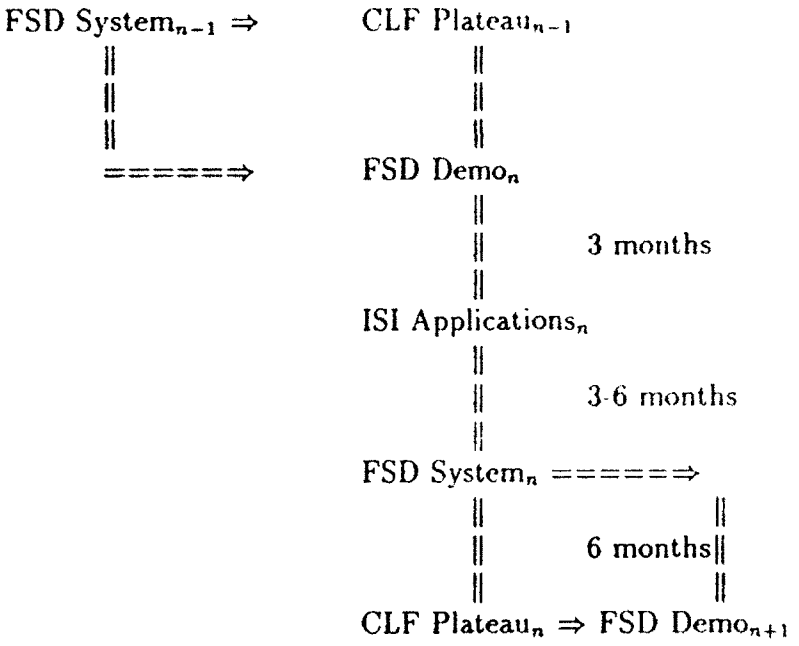

Figure 1: FSD-CLF plateau cycle

1984. A preliminary version of the CLF was in beta test by the end of 1985 and was delivered to other SC contractors in early 1986.

Hence, CLF was basically a technology transfer project, adapting well-understood aspects of the FSD system to COMMON LISP programming support. The development cycle of functionality obeyed the diagram in Figure 1. As each system feature matured in the FSD system, it was incorporated into the CLF. That CLF facility then became the foundation for the next version of the FSD system. Generally, features underwent two distinct phases. The initial "demonstration" phase (FSD Demo$_n$) was incomplete with respect to some intended functionality, but demonstrated a portion of that intended functionality as a straw man, a rapid prototype. After about three months, the demonstrated features were robust enough to endure "alpha test" in ISI applications. They then became official FSD system features (FSD System$_n$) and were "beta tested" at ISI for several months by a wider user community before being incorporated into the CLF (CLF Plateau$_n$). This plateau then became the base of the next FSD demonstration.

From the outset, the COMMON LISP Framework was intended to go through three major phases. The functionality of the initial CLF was best characterized as flexible management of program objects and tools. A preliminary version of this

portion was in beta test at the end of 1985 and was delivered to other SC contractors in early 1986. During the second phase, the tools provided in the initial system were made more reliable and robust. The third phase was to begin to incorporate the FSD technology for program specification, implementation via transformation, and reimplementation via replay of previously formalized developments on changed specifications. These phases were viewed as merely the first of many in which spinoffs from the FSD testbed are incorporated into the CLF for technology transfer to the SC contractors.

The functionality of the initial CLF was best characterized as fully flexible management of all program objects and tools. It incorporated: the relational database of program objects (modules, functions, record declarations, etc.), analysis predicates (flow relationships, call relationships, variable usage patterns) via which all tools communicate; facilities for creation, modification, destruction, retention, organization, and retrieval of these objects; the interactive standardized user interface to these facilities using a menu/window system; tools which interface to these objects including the ZMACS editor and a batch invoked tool for program- and data-flow analysis; and, the "DEVELOP" mechanism to record a history of change within the FSD environment.

The system was released for "beta test" in FY86, incorporating: a facility for converting existing programs for use in the framework; a preliminary facility for managing system releases based on the DEVELOP facility mentioned above; and, a brief manual and an excellent on-line tutorial introduction to normal usage of the system. Access to the underlying CLF facilities was provided so that developers could provide a "consistent underlying viewpoint" to the users of their facilities. CLF was ported into Common Lisp and onto other host machines.

During the second phase, the tools provided in the initial system were made more reliable and robust; new versions of the database and our interface were incorporated to increase responsiveness through heterogeneous data representation and to provide object-oriented browsing and editing.

In fact, the technical problem was redefined during the progress of the current contract, for two fundamental reasons:

- The FSD technology for program specification, transformation, and development replay did not progress rapidly enough to advantageously be incorporated in the CLF. Indeed, the technology ultimately was rejected in favor of a more *domain-specific* aproach to specification and a much more automated way of developing good implementations (See follow-on project, "*Annotations + Metaprograms*".

- The CLF was not accepted externally because it was perceived as a "monolith" which had to be accepted as a whole, rather than adopted incrementally into

existing environments containing important, entrenched functionality.

Hence, the technical problems gradually devolved to exporting the technology in a form suitable for use by other technologists. This involved porting the CLF to an open-architecture system, Unix, before we could seriously propose that others use the system components (they were written for Lisp Machines of the early 1980s). Of course, then there was considerable refining, redesigning of component interfaces, production of user manuals, etc., so they could indeed by extracted from the system and presented to external users.

3 General Methodology

3.1 Salient Features of the CLF

The COMMON LISP Framework comprises the following major components:

- AI Operating System (CLF Kernel);
- Program Management Service;
- Program Development Service.

The AI operating system kernel of the CLF provides the programmer with a fileless environment of persistent objects. Their relationships with other objects are manipulated via a set of generic operations and are used to retrieve the objects associatively. The CLF maintains the consistency of these objects and initiates automated processing for the programmer via a set of rules.

A unique feature of the CLF is that all objects manipulated by programmers are represented in this persistent object-base — specifically structured objects, like modules with components, as well as primitive program structures, like function, variable, and structure declarations. Also, relationships and objects elsewhere represented in a very *ad hoc*, diversified fashion, such as flow analysis relationships, versions, time stamps, developers, and users, are all represented uniformly in the object-base.

The kernel provides generic facilities for manipulating these programming objects as objects. In addition, special facilities have been introduced into the framework to provide programming assistance in the following areas:

- Module creation;
- Component addition;
- Importing existing files;

- Component modification and editing;
- Code installation.

Each of these facilities requires user intervention or initiation.

Especially important are the facilities for managing the consistency of program information, built using the rule-based kernel. These facilities perform the following activities:

- Automatically compile functions that are "installed" by the user;
- Allow multiple-buffer editing of the same object while maintaining correct views in each buffer;
- Automatically (re)analyze functions when they are installed;
- Automatically maintain program object ordering by load-order and view-order.

Each of these activities is managed differently in existing programming environments. Some are managed by the user only–the system provides no help for such consistency maintenance.

The CLF's Program Development Service provides automated maintenance documentation by maintaining an annotated development history. The user is responsible for providing development step annotations briefly characterizing his activity when he changes the program. The programmer may explicitly indicate substructure in his development, whereupon the system maintains his stated goal structure.

Of considerable importance is the ability of a user to indicate his plans for future programming activity, through creation of development steps called "pending steps." The user can subsequently–perhaps at a much later time–handle these development steps; the system automatically incorporates them as new steps in the existing structured history.

Not only is the development service able to recall the history of the programming activity itself, but it is linked into the persistent object-base management activity in such a way that information associated with the changes may be used for maintenance documentation and release and for version management. Thus, since the generic object-base facility is used to store the development history *itself*, one can find all the development steps affecting a particular object, as well as all the objects affected by a particular development step.

This link into the incremental saving mechanism allows the development service to provide automated distribution of program objects to users of the systems of which they are components, as well as to aid in tracking the installation of sets of changed program objects when the affecting development steps are accepted. The documentation used to describe development steps is used to document the distributed changes

to users of the system. This is a particularly interesting side-effect of our efforts to record as much as possible of the programming *process* in the machine, where the information can be analyzed and the user aided based on this analysis.

The reader may feel more comfortable with these concepts by considering the normal usage scenario in Figure 2. New users of the CLF usually have running code that they must import into the CLF environment. Often they will analyze and reorganize their original flat files into a more logical structure of nested modules using the CLF's object manipulation capabilities. After this importing is complete, the development phase is repeatedly performed. In each cycle development steps are created, augmenting the system, fixing a bug, or tuning the system. A series of modifications is made to accomplish that change. The changes are installed and tested. When the changes work properly, the development step is closed and the changes in it are distributed to the user of the modified system. All of this recording, installation, and distribution is automated.

From time to time the developer needs to reestablish his persistent object-base in a new virtual address space (e.g., when a new version of the CLF is released or when the system crashes). The distribution information automatically saved by the CLF is sufficient to automatically restore the current state of his systems and their development in the new object-base.

3.2 Conversion and Porting

The general approach taken in converting facilities in the existing CLF in order to port them onto a Unix platform was to convert base functionality, α-test it for a few weeks on the new platform, and then let the rest of the Software Sciences Division personnel β-test it to wring out the bugs. This entailed converting the "AP5" system — our virtual memory database programming language — before any of the rest. A second major problem involved the linkage between the editor, Epoch, and the Common Lisp system. On the Lisp machine implementations of the CLF from previous years, the editor was an implicit, integral part of the Lisp environment. On the Unix platform, the editor process was quite separate from Lisp, and did not communicate with other processes. Hence, another major conversion problem was setting up a remote procedure call linkage between the Common Lisp process and the editor process, through the use of Unix "sockets." Some details of the porting process are reported in the Technical Results section below.

3.3 Exporting Components

The approach to component exportation was quite simple: we identified which components could be exported, what other components they relied on — both internally

Startup Phase:
- Three ways to obtain a system:
  - **Conversion**
    - Convert existing file into module
    - Reorganize and subdivide module
    - Analyze
    - Make module into system (or Save module)
  - **Load Saved System**
    - Analyze
  - **Create New System**

Development Phase:
- Plan development
  - Make modifications
  - Test
  - Examine modifications and iterate
- Distribute modifications

Shutdown Phase: (moving to a new band)
- Save module
- Update systems in new band

Figure 2: Normal Usage Scenario

produced and externally produced, and then made them available as they were ported. This last task sometimes entailed decoupling them from their environment and usually required producing some kind of user manual or at least a reference manual for them.

The components we identified early as both suitable and desirable for exportation are described in Figure 3. Several other components arose during the course of the project which were also made available for distribution. These are all described below in the Technical Results section.

4 Technical Results

4.1 AP5

AP5 is the basis for the AI operating system provided by the CLF. In particular, it is an extension of Common Lisp providing a virtual memory data base with automatic consistency maintenance, demonic function invocation, and user-suppliable implementations for relations.

During FY87, we introduced several "specification-based" facilities to separate specification of functionality from implementation. Foremost was compiler "annotations" which instruct the compiler to choose particular representations for abstract data structures, and to optimize based on size and effort estimates for particular structures. Another such annotation was to allow POPART's ability to build and maintain grammatical structures act as relations. We also incorporated incremental flow analysis and compilation facilities.

Basically, AP5 was ready for exportation as soon as it was ported to the Unix based Common Lisp system. An important training manual was written to enhance its appeal.

4.2 Worlds

"Worlds" was a less mature system, built on top of AP5 to provide persistence for virtual database objects and relations. Worlds partition the database into layers of conceptually related data which can be composed to construct "complete" objects or relations. The mechanism is now used routinely to maintain mail and personnel data and is in experimental use in maintaining programming service data. Worlds is unique in that a world is really just a "view" of the database — rarely including all attributes of any particular object that it contains.

During FY88 the worlds system matured considerably. During FY89 we introduced a facility we built to allow users to specify a set of 'seeds' via a predicate. The

The CLF Framework itself:
  Dependencies: Common Lisp, one of Symbolics 3600, TI Explorer,
                HP Bobcat
  Interface: Extends environment of host machine
  Description: (See text)

AP5
  Dependencies: Common Lisp
  Interface: Language extension to Common Lisp
  Description: Provides virtual memory data base with automatic
               consistency maintenance, demonic function invocation,
               and user-suppliable implementations for relations.

Forms Kit
  Dependencies: Common Lisp, X-Windows, Common Objects
  Interface: Common Lisp macros for form definition; function
             invocation for display.
  Description: Generic object display mechanism with support for a
               variety of standard formats.

Popart
  Dependencies: Common Lisp
  Interface: Function calls, ZMACS macros, pseudo-LISPX macros
  Description: Language independent programming environment
               generator from language description in BNF variant.
               Provides: parser, pattern matcher, lexical analyzer,
               pretty-printer, semantic "action" routine mechanism,
               structure editor, and transformation system.

Worlds
  Dependencies: AP5, Common Lisp
  Interface: Function invocation
  Description: Object persistence maintenance mechanism based on
               conceptual aggregates of types and relations into
               "worlds".

Popart-DB
  Dependencies: AP5, Popart, Common Lisp
  Interface: Popart and AP5 Interfaces
  Description: Integrates Popart's abstract syntax declarations for
               a grammar with AP5's relation and type declarations.

Figure 3: Exportable Components

worlds mechanism then uses this predicate to start its closure algorithm to determine what belongs in the world. Most importantly, the mechanism understands the *incremental* changes to the seed set when subsequently invoked to repopulate a world that has changed. This somewhat simple enhancement will significantly decreased the overhead of the users of the persistence mechanism. We also discovered that the simultaneous commital of information from several worlds to the persistent store can be done more efficiently than the sequential commital of each. We modified the algorithm to take advantage of this opportunity.

USC graduate student Surjatini Widjojo completed a demonstration system (for her dissertation work) called "Worldbase". This system allows the definition of worlds, information aggregates which can be used to make persistent, data accumulated in the workstation. These world images can then be loaded, transformed, and merged into other workstation environments. This is especially nice work giving the project visibility in the database community.

4.3 Popart

Popart is a language independent programming environment generator that takes a language description in BNF variant and provides a parser, pattern matcher, lexical analyzer, pretty-printer, semantic "action" routine mechanism, structure editor, and transformation system. It was developed under DARPA and NSF support during the late 1970s and early 1980s. It is unique in that it provides a *concrete syntactic view* to all tool builders and users of the system, despite maitaining everything in an abstract syntax internally. This is just one aspect that makes it extremely useful for rapid prototyping of language processing facilities.

Early in 1989, the FSD project developed "Barrel", a 'Passive Virtual Database' programming architecture. It shares AP5's representation library concept with the active database architecture but requires no run-time support, and thus, no run-time overhead. This invention presented an opportunity to generalize our Popart system considerably. Particular areas of generalization were:

- Grammar-related state variables. Previously, most grammar related information was only accessible in one direction, viz from the grammar to the information. There are situations where it would be useful to access the grammar from the information. This was simply impossible without considerable reprogramming. Representing these relationships as Barrel relations gives us the flexibility of changing the representation and allowing the inverse access without changing the program at all. More important, over-restrictive constraints on uniqueness of names, productions, operators, etc., were required in the past, because these bits of information were not parameterized with respect to the

grammar in which they occur. This quarter we reparameterized these items, allowing the relaxation of the uniqueness constraints.

- Similarly, the abstract syntax used in Popart was very rigidly implemented using list structures. Barrel allowed us first to generalize the abstract syntax to include the grammar of which it is a part as an argument to relations describing it. Then differential abstract syntaxes (for different grammars) were invented.

The reprogramming of Popart to incorporate these two extensions was a major accomplishment of FY90. About 85% of the code that we intended to convert was converted and tested. This not only effected a cleaner, easier-to-modify version of Popart but allowed for additional functional enhancements[1].

During FY91, a generic facility to permit "whitespace parsing" was added to the Barrel version of Popart. The unique feature of this mechanism is its unobtrusive impact to the original syntax of the language in which the whitespace is embedded.

4.4 Luke

An important component developed in the FSD project and subsequently exported through the CLF project was LUKE (Lisp Universal Kode Elaborator), a code walking "shell" for Common Lisp and languages embedded within it. Code walkers are used for a variety of program analysis tools. Luke is a shell in the sense that it performs no useful task in its own right; it must be tailored to a particular application by providing certain methods that determine the code walking path and result computation for that application. LUKE can be used either to compute a transformation (some *image*) of a piece of code, or solely for side effect — e.g., to gather statistics about the code.

Conceptually LUKE derives extensibility by viewing the task of code walking as an example of recursive descent programming, and enabling the programmer to factor concerns about a given recursive descent application into

- the problem of *identifying* the relevant subtrees of a given tree, and
- the problem of *composing the result* for a given application.

Programmatically LUKE derives extensibility from its implementation in CLOS. This enables the user to define a new application as a *specialization* of another, similar application and to describe only the differences. The use of *mulitmethods* enables these differences to be described in terms of a "natural" syntax for Common Lisp.

[1] Although most of the reprogramming of Popart was concluded, the efficiency with which it runs was never acceptable: the appropriate Barrel annotations were not programmed. Hence, the reprogrammed Popart did not replace the existing one for most of its user community. As time permits, follow-on contracts will continue development of the Barrel version of Popart.

The standard LUKE shell maintains contextual information that can be accessed by an application, including: declarations in effect for the code being processed, enclosing lexical blocks, enclosing named definitions, and contextually imposed type requirements. LUKE provides a *template* language to ease the programming of extensions to cover user-defined macros. Templates are *compiled* into methods, not interpreted during the code walk.

4.5 AP5, Barrel, and Compilation Framework

As was mentioned above, early in 1989, the FSD project developed "Barrel", a 'Passive Virtual Database' programming architecture. It shares AP5's representation library concept with the active database architecture but requires no run-time support, and thus, no run-time overhead. In the CLF project, Barrel was extended to keep track of statistics of relation update and access for those relations using the default implementation. After using a system for a while, the user can then check to see what relations' speeds are proving to be bothersome.

At the same time, a Relational Abstraction Compilation Framework was developed in the FSD project which generalizes both the Barrel compiler and the AP5 compiler to provide a framework for compiling extensions to *arbitrary* programming languages to provide relational abstraction. Some of that work was taken up by the CLF project as the FSD project wound down. In particular, methods were added to allow more update interfaces to the abstract relations. A more elegant encapsulation of retrieval methods was also discovered, whereby despite using an iterative interface to retrieval generation, efficient single–selections and simple queries are supported.

During FY91 the framework was extended in two major ways:

- Derived relations in logic formulae were implemented. Target–language–independent algorithms are produced (consisting of neutral instructions like "generate this domain first", "test this predicate now", etc.) by the well–formed–formula compiler. Subsequently, target–language–specific source code generators and cost estimators are invoked by the framework.

- A compositional specification language was designed and implemented, allowing users to describe how to represent a relation by composing different indexing regimes from a library of predefined representations.

4.6 Forms Kit

"Forms Kit" is a generic object display mechanism with support for a variety of standard layout formats. It was another component whose functionality was obviously

ready for exportation, but whose porting onto Unix platforms was problematical because of the discrepancies between X-windows and the idiosyncratic window systems of the Lisp Machines. In addition, it made several assumptions about the existence of AP5, which made it inappropriate for exportation to the research community — there was no reason for us to bundle these two potential products together[2].

During FY88 we completed the design, implementation, and documentation of a specification-based user interface, independent of AP5. Specifications control content, layout, graphic characteristics, and user interaction with displayed data. We also particularized this tool (by adding a new abstract interface) for use within CLF so that the data is obtained from the AP5 database and the form is kept consistent with changes in the database.

During FY89 the Forms Kit was enhanced in the following ways:

- The performance of our Screen Updater, which keeps views up-to-date with the objectbase, was improved with the implementation of a new algorithm based on the observation that the number of distinct relations modified in a typical objectbase update is quite small, whereas the number of interface units whose contents are sensitive to the objectbase may be large.

- We improved the update speed for small form changes by BITBLTing information that has not changed into new locations, rather than recomputing it as is our current practice.

- The Forms Kit system was factored into the subset necessary to create and present forms and the subset controlling user interactions, so that now the system builders can use the underlying CLX mechanisms for this control. Forms Kit will only deal with layout and presentation in the future.

- We extended documentation of the generic functions in the Forms Kit into a manual using the Interleaf System.

At this point the Forms Kit system had been factored into the subset necessary to create and present forms and the subset controlling user interactions, so that now the system builders can use the underlying CLX mechanisms for this control. Forms Kit only deals with layout and presentation now. Documentation of our Forms Kit user interface facility was updated to reflect this factoring. This completed the development of the Forms Kit facilities.

Some later additions to Forms Kit functionality included enhancing the interface package to allow the substructure of a form to be modified after it has been created and the X11 Inter-client Communication Convention Standards (ICCS) was adopted

[2]This is also one reason we implemented Popart using Barrel, rather than AP5.

in the Forms Kit implementation, allowing more effective communication between forms and X11 window managers. Functionality can now react more gracefully to mouse clicks and other gestures.

4.7 Browser

During FY90 we extended the AP5 Browser to present information in forms tailored to the data being traversed. This mechanism simply allows one to traverse the virtual database from the terminal, seeing all relations with other objects reachable from the "current" object. This facility is particularly important for HP Bobcat machine usage because of the general inability to point at items and have a mouse interface that understands how to copy them into other windows (as we had become accustomed to on the Symbolics and TI Lisp Machines). Hence, this facility — coupled with the EMACS interface — made Bobcats viable alternatives to the aging Lisp machines, by providing an interface with a 'hyptertext feel.'

4.8 Conversion/Porting

During FY89 AP5, POPART, and LUKE (code walker) were ported to Unix workstations (running Allegro Common Lisp), and our browsing/viewing mechanism (Forms Kit) was ported to X11/CLX. Conversion of the CLF to work on HP Bobcat machines continued. Another major package converted this fiscal year was the persistence mechanism, 'worlds'. In addition, the existing CLF source code was restructured into 10 major subsystems with known dependency links (from the simple-minded linear sequence of many more modules that used to constitute its structure).

During FY90 conversion of the CLF to work on HP Bobcat machines continued. The large 'programming service' modules were converted. Because of virtual memory thrashing problems, conversion of the 'program development' modules was delayed, but ultimately, additional memory to aided the situation, and the development service was converted. This concluded the conversion of the CLF, but the system remained unusable, awaiting conversion of the editor GNU (EMACS) on the HPs to act as a server. This editor interface was the final missing piece of functionality preventing our daily use of CLF on the HP machines.

During FY90 we also converted Popart to Barrel and ported our 'AP5 Browser.' Previously, AP5 had been converted to run under Allegro's version of Common Lisp. Toward the end of the fiscal year we converted AP5 to work within Lucid Common Lisp as well. Unfortunately, AP5 stresses every Common Lisp implementation it has been converted to, not by using non-Common Lisp features, but by extensive use of the more complex features (mostly having to do with closures). Hence, every conversion has entailed some implementation specific "work-arounds" to stear clear

of implementation bugs. This one only entailed implementing non-Common Lisp functionality.

We ported AP5 to the Lucid Common Lisp implementation on the HPs, partly to test the performance of that platform. The performance of the ported CLF was instrumented and measured under different memory management configurations within Allegro Common Lisp.

During FY91 the port of CLF to Lucid was completed and α-testing of the CLF system began in earnest by the original designers of the kernel facilities. Later the CLF Allegro port was given a thorough beating through β-testing by nondevelopers. The normal intensive debugging pains were incurred for a period of several weeks.

4.9 Protocols

During FY90 a new facility was developed which allows remote use of our AP5 virtual memory data base by a client machine. All relational access from this client machine is then via a remote procedure call facility. In effect, this allows a client to treat AP5 as a traditional relational database. The technical challenges here have to do with maintaining surrogate relations and objects to track the state of the other machine's view of the server's state.

We also completed the interface design between the CLF system and GNU (EMACS). The design calls for speeding up communication by replacing the use of files for interprocess communication with uses of Unix pipes and C foreign function calls for reading and writing. Toward the end of the fiscal year, CLF was converted to use a new interprocess communication protocol, sockets, instead of pipes. Using this protocol, inter–machine communication is possible, as well as the local editor/Lisp process communication that it replaced. That completed the conversion to GNU and the port of CLF to the HP platforms.

During FY91 a prototype local area network "program development server" was designed and demonstrated wherein program objects are assigned persistent identifiers and shared between several workstations. In addition, our optimistic sharing support system for program developments was propagated into the design of this server.

The conversion of AP5 to Lucid was finished. We were pleasantly surprised that we were able to obtain a significant performance improvement over the Allegro implementation, due to Lucid's facility for user-advice to control the number and sizes of ephemeral storage areas for garbage collection. Additionally, Lucid promised to support our eventual RISC-based HP platform. Hence, we chose Lucid as our Common Lisp base for further development on CLF and began to port the remainder (bulk) of CLF to Lucid.

Finally, The CLF was converted from using the GNU editor to the Epoch editor. This had fairly minor effects: allocating a single I/O buffer per process became possi-

ble, and "I/O state indicators" can be used to indicate a process' state. Most important is that Epoch enabled making a "Hypertext" facility available. We programmed the ability to have mouse actions available for CLF objects printed to ordinary text buffers.

4.10 EGS

During FY89 we made progress on a version of the EGS compiler (our specification language subset related to database programming) using our Popart transformational mechanisms called 'syntax-directed experts.' Although development of this language was ultimately dropped, the subset was later explored in the ARIES project within our division.

4.11 Training

A training manual was created for AP5, greatly facilitating its transfer to Arthur Anderson. The creation of simplified syntax for functional use of relations also helped. A manual describing our persistence mechanism, 'worlds' was updated thoroughly. The manual describing our Common Lisp code-walker, 'Luke,' was distributed.

5 Important Findings and Conclusions

5.1 Ph.D. Degrees Awarded

Surjatini Widjojo and Ed Ipser each received the Ph.D. degree in Computer Science from the University of Southern California. David Wile was the committee chairman on both committees. Both were supported by the CLF project during their tenure as graduate students.

5.2 Collaborations

During FY88 significant progress was made towards designing a proposal for 'Higher Order Abstract Syntax:' a concensus building activity in the DARPA community. The attempt here was to achieve agreement between DARPA contractors on a standard interface for grammar-based systems so that components from different sites could interoperate. The major accomplishment here was to propose a non-intrusive version of the higher order abstract syntax-an abstract interface to *it* is provided by the designer of the (simple) abstract syntax. This way the representation of the HOAS

does not have to dominate systems, but can rather coexist with (possibly) many representations of a program. Unfortunately, no implementation of the concepts (e.g. interoperating with other universities) occurred.

In order to understand the impact of an open architecture, Unix-style programming environment on CLF, and in order to gain leverage from the use of others' components, we began talking with Tom Cheatham, Mike Karr, and Glen Holloway of Software Options to understand how their E-L language and environment could interface with ours. We developed an agenda of repsonsibilities for both groups. During FY90 we converted the 'worlds' facility (our persistence mechanism) to use their 'remote lock' facility, rather than those provided by the file system, as was our practice. In addition we experimented with Software Options' "artifacts" package with an eye toward integrating its coarse-grained persistence management with AP5s fine-grained object management in the virtual memory database.

Also during FY90 we discussed with Lucid Corp. use of some of the features of their Cadillac system. In particular, they have developed a protocol for talking with editors that allows communication of structural modifications to buffers, and have converted GNU Emacs to use this protocol.

During FY90 we intended to establish a communication protocol between our Popart System's transformational semantics package and UC Berkeley's Pan system for editing grammatically structured objects. We believe that both systems have been designed flexibly enough to permit the appropriate "impedence matching" to the other's abstract syntax and grammar conventions. Pan's grammars are more limited than those accepted by Popart, so some validity checking will be necessary in the long run. Unfortunately, this collaboration was not really consumated in the end.

6 Significant Hardware Development

None.

7 Special Comments

None.

8 Implications for Further Research

The CLF is in daily use in ISI's Software Sciences Division: for several researchers it is the only system used for all computation and administration needs. It forms a

solid testbed for development of other research ideas[3].

Moreover, having ported and subsequently exported several pieces of the system has enabled collaborations both within the division and institute, and beyond. The ARIES project has used several of the foundational pieces — AP5, Popart, Popart/DB, Kodewalker — to produce its system for obtaining and reasoning about program requirements. The Popart mechanisms have been used in Knowledge Base representation systems transformation within another division.

Furthermore, collaborations with other DARPA contractors has been facilitated. Subsequent integration of some of the Software Options artifacts with the CLF to provide a "process programming" base have been especially interesting.

CLF indeed has influenced the development of research programming environments — e.g. Marvel on Unix by Gail Kaiser — and the development of industry products, such as Cadillac by Lucid. Virtually every modern programming environment has adopted some of the concepts for an integrated programming environment that CLF provides.

Development and research on several of the products of the CLF continues on new projects now: the Popart work is continued on the Annotations + Metaprograms project and the Relational Abstraction Compiler Framework continues on the Relational Abstraction project in our division. These are both playing key roles in the emerging technology of the DARPA sponsored Domain Specific Software Architectures program.

9 Major Publications

[All89] Dennis Allard. *Worlds Manual*, May 1989.

[AW89] D. G. Allard and D. S. Wile. Aggregation, persistence, and identity in Worlds. In *Workshop on Persistent Object Systems, University of Newcastle, Australia*, January 1989.

[Fea91] M.S. Feather. Transformational implementation of historical reference. In B. Möller, editor, *Constructing Programs from Specifications*, pages 225-242. North-Holland, 1991. Proceedings of the IFIP TC2/WG 2.1 Working Conference on Constructing Programs from Specifications, Pacific Grove, CA, USA, 13-16 May 1991.

[3]The unfortunate lateness of this report at least allows us to validate these claims for success in influencing subsequent research!

[Gol89] Neil Goldman. Code walking and recursive descent: A generic approach. In *Proceedings of the Second CLOS Users and Implementors Workshop, New Orleans*, October 1989.

[Gol91] Neil Goldman. *CLF Manual*, September 1991.

[HWW90] R. Hull, S. Widjojo, and D. S. Wile. *A Specificational Approach to Database Transformation*. Morgan-Kaufmann, December 1990. editors: A. Dearle and G. Shaw and S. Zdonik.

[HWWY91] R. Hull, S. Widjojo, D. Wile, and M. Yoshikawa. On data restructuring and merging with object identity. *IEEE Data Engineering Bulletin, Special Issue on Theoretical Foundations of Object-Oriented Database Systems*, 14(2), June 1991.

[IJW90] Edward A. Ipser, Jr., Dean Jacobs, and David S. Wile. A multi-formalism specification environment. In *Proceedings of the Fourth International Conference on Software Development Environments, Irvine, California*, December 1990.

[Ips90] Edward A. Ipser, Jr. *Toward A Multi-Formalism Specification Environment*. PhD thesis, University of Southern California, Computer Science Department, 1990. University of Southern California Information Sciences Institute.

[Mey89] Jay Meyers. *AP5 Training Manual*, April 1989.

[Mil89a] Brent Miller. *Forms Kit Kernel Manual*, 1989.

[Mil89b] Brent Miller. *ISI Extensions Manual*, 1989.

[Nar91] K. Narayanaswamy. What exactly is a variant? In *Proceedings of the 3rd International Workshop on Software Configuration Management*. ACM, June 1991.

[Wid90] S. Widjojo. *WorldBase: A Distributed Information Sharing System*. PhD thesis, Computer Science Department, University of Southern California, Los Angeles, CA, 1990.

[Wil89] David S. Wile. Carving up SCAD databases. In *Proceedings of a Workshop on Software CAD Databases, Napa, California*, February 1989.

[WWH90] S. Widjojo, D. S. Wile, and R. Hull. WorldBase: A New Approach to Sharing Distributed Information. Technical report, USC/Information Sciences Institute, February 1990.